

Adabas Design

Database systems often involve complex data structures and data handling procedures that can be designed and used only by persons with extensive knowledge and experience. Adabas has a remarkably simple structure by comparison, yet it provides significant advantages for operational efficiency, ease of design, definition, and database evolution.

This chapter covers the following topics:

- Adabas Entities
 - Database Components
 - Database Files
 - Records and Field Definitions
-

Adabas Entities

In Adabas, a "field" is the smallest logical unit of information (e.g., current salary) that may be defined and referenced by the user. A "record" is a collection of related fields that make up a complete unit of information (e.g., all the payroll data for a single employee). A "file" is a group of related records that have the same format (with some exceptions; see *Multiple Record Types in One File*). A "database" is a group of related files.

Adabas Limits

The table below shows the maximum number that mainframe Adabas supports for each entity:

Entity	Maximum
Databases	65,535
Blocks per database	2,147,483,646 using 4-byte RABNs
Files per database	the lower of 5,000 or the Associator block size minus one
Records per file	4,294,967,294 using 4-byte ISNs
Fields per record	926
Uncompressed record length	depends on the operating system
Compressed record length	Data Storage block size

Adabas Space Management

The disk storage space allocated to a single Adabas database is segmented into *logical* Adabas files. A certain part of the overall space within the database is allocated to each logical file. When the space is filled with records from the file, Adabas automatically allocates more space to the file from the common free space pool. This dynamic space allocation, together with the dynamic recovery of released space,

allows Adabas databases to run without intervention for long periods of time.

The distribution of database space across disk drives can be controlled by "physically" segmenting it into multiple independent datasets. When all physical database space is filled, more datasets can be allocated dynamically, or the size of existing datasets can be increased so that new physical files can be loaded without reorganizing the entire database.

Database Components

To support the separation of data and access structures, the Adabas nucleus uses three database components:

- Data Storage for compressed data
- Associator for data management and retrieval
- Work, a scratch area for complex search criteria, etc.

Data Storage

Data Storage is divided into "blocks", each identified by a 3- or 4-byte relative Adabas block number or "RABN" that identifies the block's physical location relative to the beginning of the component. Data Storage blocks contain one or more physical records and a padding area to absorb the expansion of records in the block.

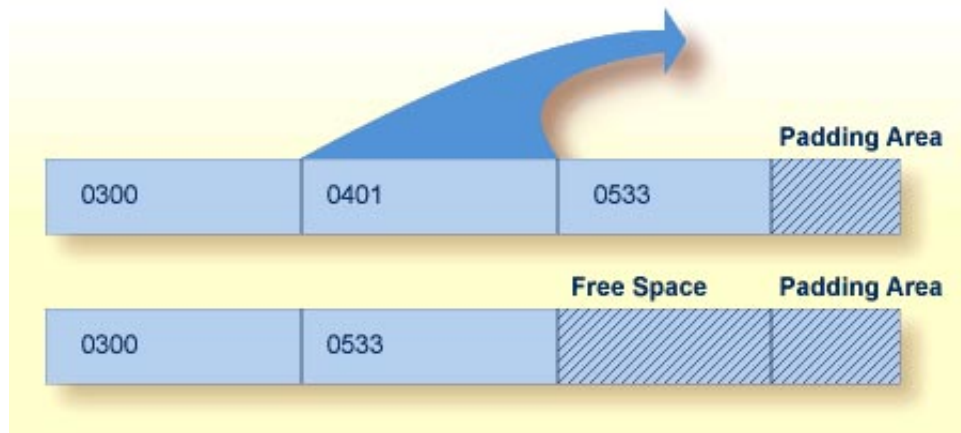
A logical identifier stored in the first four bytes of each physical record is the only control information stored in the data block. This internal sequence number or "ISN" uniquely identifies each record and never changes. When a record is added, it is assigned an ISN equal to the highest existing ISN plus one. When a record is deleted, its ISN is reused only if you instruct Adabas to do so. Reusing ISNs reduces system overhead during some searches and is recommended for files with records that are frequently added and deleted.

For each file, between 1-90 percent (default 10%) of each block can be allocated as padding based on the amount and type of updating expected. This reserved space permits records to expand without migrating to another block and thus helps to minimize system overhead.

RABN	Records (identified by ISNs)			Padding Area
1	0071	0595	0221	
2	0222	0991	2021	
3	0300	0401	0532	

Free Space and Space Reusage

If records become too large for their blocks, they migrate to new locations. When a record migrates or is deleted, free space is opened in the data block between the last record and the padding area. The following figure shows free space created when the record with ISN 0401 becomes too large for the block and migrates to another block:



You can instruct Adabas to reuse free space. Reusing space saves computer time, since Adabas then reads fewer physical blocks during searches. It is recommended for all files.

Compression

Data compression significantly reduces the amount of storage required. It also permits the transmission of more information per physical transfer, resulting in greater I/O efficiency.

Adabas retains data records in compressed form. Four compression options are supported:

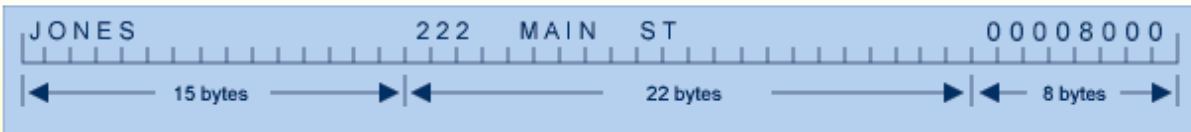
- default compression;
- null suppression;
- fixed format; and
- forward or "prefix" index compression.

The first three options define and execute compression at the field level; the fourth option can be implemented at the file or the database level, in which case specific files can be set differently; the file-level setting overrides the database setting. The null suppression and fixed format options are added as field options and are discussed in *Data Compression Options FI and NU*. The forward index compression option is set using the ADALOD utility and can be changed using the ADAORD utility.

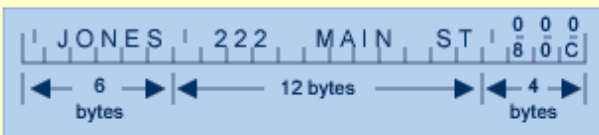
"Default compression" deletes trailing blanks in alphanumeric fields and leading zeros in binary fields. An inclusive length byte (ILB) at the beginning of the field indicates the total number of stored bytes, including the ILB. Thus, if "Susan" is entered in a "first-name" field defined with a 20-character length and default compression, its stored size will be six bytes: five bytes for the letters of the name, plus one byte for the ILB. In addition, empty fields in a record are not stored; an empty field is replaced by a one-byte empty field counter (EFC). Adabas can store up to 63 contiguous empty fields in a single hexadecimal byte.

Many Adabas files require only 50% to 60% of the space used for the raw data. Even with the addition of approximately 25% for the access structures stored in the Associator, Adabas storage requirements are still less than those required for traditional file storage or for DBMSs that do not use data compression.

Data Before Compression (45 Bytes)



Adabas Compressed Data (22 Bytes)



Forward (or 'front' or 'prefix') index compression removes redundant prefix information from index values. Within one index block, the first value is stored in full length. For all subsequent values, the prefix that is common with the predecessor is compressed. An index value is represented by

<l,p,value>

-where

p	is the number of bytes that are identical to the prefix of the preceding value.
l	is the exclusive length of the remaining value including the p-byte.

For example:

Before Compression	After Compression
ABCDE	6 0 ABCDE
ABCDEF	2 5 F
ABCGGG	4 3 GGG
ABCGGH	2 5 H

The decision to compress index values is based on the similarity of index values and the size of the file:

- the more similar the index values, the better the compression results.
- small files are not good candidates because the absolute amount of space saved would be small whereas large files are good candidates for index compression.

Even in a worst case scenario where the index values for a file do not compress well, a compressed index will not require more index blocks than an uncompressed index.

Associator

The Associator is an organizational unit used for storing the structures required to access data in Data Storage. It contains

- a control block for the database as a whole and control blocks for each file;
- all tables needed to control and maintain the database including a field definition table or "FDT" (see *Records and Field Definitions*) for each file and coupling lists for physically coupled files (see *Coupled Files*);
- an inverted list for each descriptor in each file of the database and an address converter for each file.

Inverted Lists

An inverted list, which is used to resolve Adabas search commands and read records in logical sequence, is built and maintained for each field in an Adabas file that is designated as a key field or "descriptor" (see *Descriptor Options DE, UQ, and XI*). It is called an "inverted" list because it is organized by descriptor value rather than by ISN. The list comprises the normal index (NI) and as many as 14 upper indexes (UI).

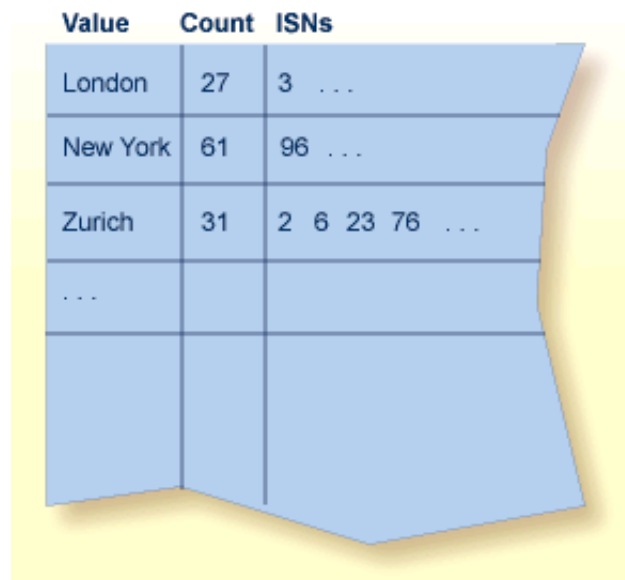
The normal index (NI) of the inverted list for a particular descriptor has an entry for each value. The entry contains the value itself, the number of records in which the value occurs, and the ISNs of those records.

To increase search efficiency, upper index (UI) levels are automatically created by Adabas as required, each level to manage the next lower level index. The first level UI, like the NI it manages, contains entries for only one descriptor in each index block. All other UI levels contain entries for all descriptors in each index block. UIs require a minimal amount of space: two blocks is the minimum.

Note:

The Adabas direct access method (ADAM) facility permits the retrieval of records directly from Data Storage without accessing the inverted lists. The Data Storage block number in which a record is located is calculated using a randomizing algorithm based on the ADAM key of the record. The use of ADAM is completely transparent to application programs and query and report writer facilities. See *Random Access Using the Adabas Direct Access Method (ADAM)* for more information.

The following figure shows a typical normal index for the descriptor "city" in a customer file.



Value	Count	ISNs
London	27	3 ...
New York	61	96 ...
Zurich	31	2 6 23 76 ...
...		

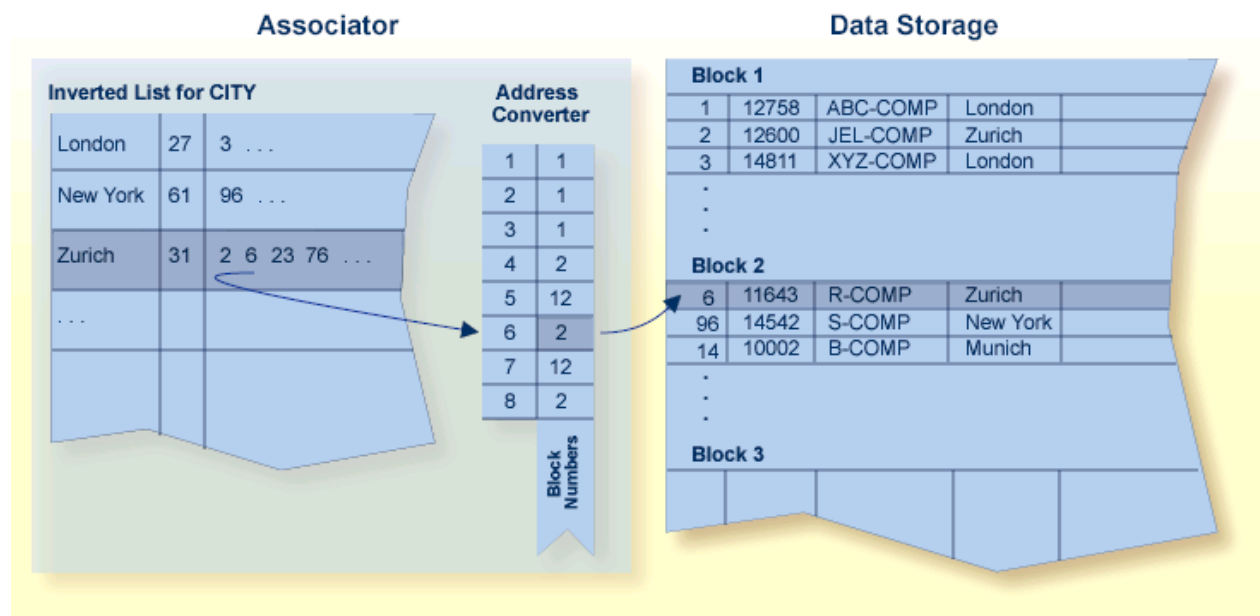
The example indicates that there are 31 records with the "city" Zurich (the ISNs of these records are 2,6,23,76...).

Address Converter

The address converter determines the physical location of a record. It is an index that maps the logical identifier of a record (that is, the ISN) to the relative Adabas block number (RABN) of the Data Storage block where the record is stored.

The address converter contains a list of RABNs in ISN order. Only the RABNs are actually stored in the address converter; the ISNs are identified by their relative position.

The following figure shows the relationship between an inverted list, the address converter, and Data Storage. For example, to determine the physical location of the record whose ISN is 6, Adabas uses the ISN as an index into the address converter. The sixth entry in the address converter is 2. Therefore, ISN 6 is located in physical block 2 in Data Storage for this file.



When a record moves or is deleted, Adabas updates the address converter automatically and transparently.

Since the ISN for a record never changes, and its physical block address is stored only in the address converter entry, the record itself may be moved in Data Storage with only one update to the address converter required and with no extension to the access path of the record.

Even if a record has many descriptors defined, the inverted list for each descriptor need not be modified because it contains ISNs.

This process explains how Adabas is able to perform simple and complex searches quickly and efficiently without storing pointer information in Data Storage.

Work

The Work area stores information in four parts:

Part	Stores ...
1	data protection information required by the routines for autorestart and autobackout. See <i>Backout</i> , <i>Recovery</i> , and <i>Restart</i> for more information.
2	intermediate results (ISN lists) of search commands.
3	final results (ISN lists) of search commands.
4	data related to two-phase commit processing.

Other Components

Sort and Temp Areas

Certain Adabas utilities (ADAINV, ADALOD) require two additional datasets, sort and temp, for sorting and intermediate storage of data. Certain functions of other utilities require the temp dataset for intermediate storage.

The size of the temp and sort datasets varies according to the utility function to be executed. These datasets can be allocated during the job and then released, or permanent datasets can be allocated and reused.

Logs

Adabas uses the following optional logs:

- The "command log" (CLOG) records information from the control block of each Adabas command that is issued. The CLOG provides an audit trail and can be used for debugging and for monitoring the use of resources. Single, dual, or multiple (2-8) datasets can be used (multiple datasets are recommended).
- The "protection log" (PLOG) records before- and after-images of records and other elements when changes are made to the database. It is used to recover the database (up to the last completed transaction or "ET") after restart. Single, dual, or multiple (2-8) datasets can be used (multiple datasets are recommended).
- The "recovery log" (RLOG) records additional information that the Adabas Recovery Aid uses to construct a recovery job stream. See the ADARAI utility discussion for more information.

Database Files

Each database contains system files and data files. A data file is generally created for each record structure required; that is, for each set of related fields identified.

Files are loaded into the database using the ADALOD utility. A file number must be unique in the database and not greater than the maximum file number defined for the database in the MAXFILES parameter. For a checkpoint, security, system file, or physically coupled file, the number cannot be greater than 255; other files including a trigger file can have two-byte file numbers. File numbers are assigned by the user in any sequence.

System Files

Adabas uses certain files to store system information. Using the ADALOD utility's FILE parameter, you can identify an Adabas *system* file as one of the following:

CHECKPOINT	Adabas checkpoint file
SECURITY	Adabas security file
SYSFILE	Adabas system file
TRIGGER	Adabas trigger file

Coupled Files

File coupling allows you to select, using a single search command, records from one file that are related (coupled) to records containing specified values in a second file.

Physical Coupling

Any two files with file numbers 255 or lower may be physically coupled if a common "descriptor" (see *Descriptor Options DE, UQ, and XI*) with identical format and length definitions is present in both files. A single file may be coupled with up to 18 other files, but only one coupling relationship may exist between any two files at any one time. A file may not be coupled to itself.

When files are coupled, coupling lists are created in the Associator for each file being coupled. File coupling is bidirectional rather than hierarchical in that two coupling lists are created for each coupling relationship with each list containing the ISNs that are coupled to the other file.

Once the physical coupling lists have been created, any key field in either file may be used within a search criteria.

Physical coupling may add a considerable amount of overhead if the files involved are frequently updated. The coupling lists must be updated if a record in either of the files is added or deleted, or if the descriptor used as the basis for the coupling is updated in either file.

Physical coupling may be useful for information retrieval systems in which

- files seldom change;
- the additional overhead of the coupling lists is insignificant compared with the increased ease of formulating queries; or
- files are small and primarily query-oriented.

Logical or "Soft" Coupling

Multiple files may also be queried by specifying the field to be used for interfile linkage in the search criteria. Adabas then performs all necessary search, read, and internal list matching operations.

This technique is called logical or "soft" coupling because it does not require the files to be physically coupled. Although logical coupling requires read commands, it is normally more efficient because it avoids the increased overhead of coupling lists.

Structuring Files to Enhance Performance

An Adabas database with one file for each record type supports any application functions required of it and is the easiest to manipulate for interactive queries, but it may not yield the best performance:

- As the number of Adabas files increases, the number of Adabas calls increases. Each Adabas call requires interpretation, validation and, in multiuser mode, supervisor call (SVC) and queuing overhead.
- In addition to the input/output (I/O) operations necessary for accessing at least one index, address converter, and Data Storage block from each file, the "one file per record type" structure requires buffer pool space. If sufficient buffer space is not available, blocks are overwritten that may be

needed for a later request.

The number of Adabas files used by critical programs can be reduced by

- using multiple-value fields and periodic groups (see *Field Levels*);
- linking physical files into a single logical (expanded) file;
- including more than one type of record in an Adabas file;
- including records for more than one category of user in an Adabas (multiclient) file; and
- controlling data duplication and the resulting high resource usage.

Expanded Files

If you have a large number of records of a single type, you may need to spread the records over multiple physical files.

To reduce the number of files accessed, Adabas allows you to link multiple physical files containing records of the same format together as a single logical file. This file structure is called an "expanded file" and the physical files comprising it are the "component files". An expanded file can comprise up to 128 component files, each with a unique range of logical ISNs. An expanded file cannot exceed 4,294,967,294 records.

Note:

Since Adabas now supports larger file sizes and a greater number of Adabas physical files and databases, the need for expanded files has, in most cases, been removed.

Although an application program addresses the logical file (the address of the file is the number of the expanded file's base component or "anchor" file), Adabas selects the correct component file based on the data in a field defined as the "criterion" field. The data in this field has characteristics unique to records in only one component file. When an application updates the expanded file, Adabas looks at the data in the criterion field in the record to be written to determine which component file to update. When reading expanded file data, Adabas uses the logical ISN as the key to finding the correct component file.

Multiple Record Types in One File

Multiple record types can be defined within a single physical record; each record type is a logical record composed of a subset of the fields defined for the file. Fields that do not belong to a given type are null-suppressed.

Record types can be identified to Adabas by

- defining a record type field with values to differentiate one type from another; or
- using values of an existing field to differentiate type; for example, to differentiate two types, a value of zero for a field common to both types might identify one type and any nonzero value for the same field might identify the other type.

Multiclient Files

Records for multiple users or groups of users can be stored in a single Adabas physical file defined as "multiclient". The multiclient feature divides the physical file into multiple logical files by attaching an internal owner ID to each record.

The owner ID is assigned to a user ID. A user ID can have only one owner ID, but an owner ID can belong to more than one user. Each user can access only the subset of records that is associated with the user's owner ID.

Note:

For any installed external security package such as RACF, CA-ACF2, or CA-Top Secret, a user is still identified by either Natural ETID or LOGON ID.

All database requests to multiclient files are handled by the Adabas nucleus.

Controlled Data Redundancy

"Physical" redundancy increases storage requirements but may also enhance performance and decrease complexity. For example, if a database stores customer and order information in a customer-orders file and product descriptions in an inventory file, and a program that generates invoices requires product descriptions in addition to customer-order data, it might enhance performance to store a duplicate copy of the product descriptions in the customer-orders file.

"Logical" redundancy also increases storage demands while decreasing complexity. It involves storing in one file the results of a process on data in another file; thus, the duplicate data is implied by the content of another file, rather than being physically stored in two places.

Physical and logical redundancy cause update programs to run more slowly. The duplicate updates required when changes in one file affect records in another file may degrade performance severely. Redundancy should be used only for static data or data that is updated rarely. You can control data redundancy by using multiple-value fields, periodic groups, and multiple record types within a file.

Records and Field Definitions

In Adabas, the record structure and the content of each field in a physical file are described in a field definition table or "FDT", which is stored in the Associator. There is one FDT for each database file. The FDT is used by Adabas during the execution of Adabas commands to determine the logical structure and characteristics of any given field (or group) in the file.

The FDT lists the fields of the file in physical record order; provides a "quick index" to the file's records; and defines the file's fields, sub/superfields, and descriptors including collation, sub-/super-/hyper- and phonetic. A minimum of one and a maximum of 926 field definitions may be specified.

Information about each field includes the level, name, length, format, options, and special field and descriptor attributes.

FIELD DESCRIPTION TABLE							
LEVEL	I	I	I	I	I	I	I
	I	I	I	I	I	I	I
	NAME	LENGTH	FORMAT	OPTIONS		PARENT OF	
-----	I	I	I	I	I	I	I
1	I	AA	I	8	I	A	I DE,UQ
1	I	AB	I		I		
2	I	AC	I	20	I	A	I NU
2	I	AE	I	20	I	A	I DE
2	I	AD	I	20	I	A	I NU
1	I	AF	I	1	I	A	I FI
1	I	AG	I	1	I	A	I FI
1	I	AH	I	6	I	U	I DE
1	I	A2	I		I		
1	I	AO	I	6	I	A	I DE
1	I	AQ	I		I		I PE
2	I	AR	I	3	I	A	I NU
2	I	AS	I	5	I	P	I NU
1	I	A3	I		I		
2	I	AU	I	2	I	U	I
2	I	AV	I	2	I	U	I NU

Record Structure

The order of the fields listed in the FDT determines the structure of the record and the efficiency of retrieval. The following factors should be considered when ordering fields:

- Fields that will be accessed frequently should be ordered first in the FDT. This technique reduces CPU time because Adabas does not have to read the whole record when retrieving a field.
- Fields that will frequently be accessed together should be assigned to a "group" field.
- Fields that will *always* be accessed together should be defined as a single field. This technique may inhibit compression and query language use; however, it decreases processing time by providing more efficient internal processing and shorter format buffers.
- If appropriate, fields that will frequently be empty should be ordered together in the FDT and set to use default compression or null suppression.
- Numeric fields should be loaded in the format in which they will be used most often.

Field Levels

When two or more consecutive fields in the FDT are frequently accessed together, you can reference them together by defining a group field. Other than its level and Adabas short name, a group field has no attributes defined. It immediately precedes its member fields in the FDT. A higher field "level" number is used to assign the member fields to the group field. Adabas supports up to seven field levels. User programs can access each member field individually, or all member fields together by referencing the group field.

For example, in the illustration of the Filed Definition Table in the section *Records and Field Definitions*, field AB is defined as a group field and assigned to level 1. Fields AC, AE, and AD are assigned to level 2, indicating that they belong to group field AB. The next field, AF, is assigned to level 1, indicating that

it is not part of the AB group. User programs can access AC, AE, and AD individually, or together by referencing the group field AB.

A group field can be assigned as a "periodic" group field if it is comprised of fields that can have more than one value (for example, group field AQ in the figure.

Field Names

A field is identified to Adabas by a two-character Adabas "short" name that must begin with an alphabetic character and can be followed by a numeral or letter (the combinations E0-E9 are reserved and special characters are not allowed) and must be unique within a file. Adabas assigns short names to fields automatically, although you can choose to assign them yourself. Adabas uses the short names internally and actually accesses fields by their short names.

Field Length and Data Format

Field values are fixed or variable in length and can be in alphanumeric, binary, fixed-point, floating-point, packed/unpacked decimal, or wide character formats.

The length (expressed in bytes) and format (expressed as a one-character code) of a field define the standards (defaults) to be used by Adabas during command processing. They are used when the field is read/updated unless the user specifies an override.

If standard length is zero for a field, the field is assumed to be a variable-length field. Standard format must be specified for a field. The format specified determines the type of default compression to be performed on the field.

The maximum field lengths that may be specified depend on the "format" value:

Format	Format Description	Maximum Length
A	Alphanumeric (left-justified): see also the long alphanumeric (LA) option in <i>Long Alpha Option LA</i>	253 bytes
B	Binary (right-justified, unsigned/positive)	126 bytes
F	Fixed point (right-justified, signed, positive value in normal form; negative value in two's complement form)	4 bytes (always exactly 2 or 4 bytes)
G	Floating point (normalized form, signed)	8 bytes (always exactly 4 or 8 bytes)
P	Packed decimal (right-justified, signed)	15 bytes
U	Unpacked decimal (right-justified, signed)	29 bytes
W	Wide character (left-justified): see also the long alphanumeric (LA) option in <i>Long Alpha Option LA</i>	253 bytes

Field Options

Field options are specified using two-character codes, which may be specified in any order, separated by a comma.

Code	Option	See Section
DE	Field is to be a descriptor (key).	<i>Descriptor Options DE, UQ, and XI</i>
FI	Field is to have a fixed storage length; values are stored without an internal length byte, are not compressed, and cannot be longer than the defined field length.	<i>Data Compression Options FI and NU</i>
LA	An alphanumeric or wide-character, variable-length field may contain a value up to 16,381 bytes long.	<i>Long Alpha Option LA</i>
MU	Field may contain up to 191 values in a single record.	<i>MU and PE Options and Field Types</i>
NC	Field may contain a null value that satisfies the SQL interpretation of a field having no value; that is, the field's value is not defined (not counted).	<i>SQL Compatibility Options NC and NN</i>
NN	Field defined with NC option must always have a value defined; it cannot contain an SQL null (not null).	<i>SQL Compatibility Options NC and NN</i>
NU	Null values occurring in the field are to be suppressed.	<i>Data Compression Options FI and NU</i>
NV	An alphanumeric or wide-character field is to be processed in the record buffer without being converted.	<i>Encoding Conversion Option NV</i>
PE	This group field is to define consecutive fields (which may include one or more MU fields) in the FDT that repeat together (up to 191 times) in a record.	<i>MU and PE Options and Field Types</i>
UQ	Field is to be a unique descriptor; that is, for each record in the file, the descriptor must have a different value.	<i>Descriptor Options DE, UQ, and XI</i>
XI	For this field, the occurrence (index) number is to be excluded from the unique descriptor (UQ) option set for a periodic group (PE).	<i>Descriptor Options DE, UQ, and XI</i>

Descriptor Options DE, UQ, and XI

A "descriptor" is a search key. The DE option indicates that the field is to be a descriptor. The UQ option can only be specified if DE is also specified; it indicates that the DE field is to have a different (i.e., unique) value for each record in the file. Entries are made in the Associator's inverted list for DE fields, adding disk space and processing overhead requirements.

Any field can be used within a selection criterion. When a field that is used extensively as a search criterion is defined as a descriptor (key), the selection process is considerably faster since Adabas is able to access the descriptor's values directly from the inverted list without reading any records from Data Storage.

A descriptor field can be used as a sort key in a search command, as a way of controlling a logical sequential read process (ascending or descending values), or as the basis for file coupling.

Any field and any number of fields in a file can be defined as descriptors. When a multiple-value field or a field in a periodic group is defined as a descriptor, multiple key values are generated for the record. Key searches may be limited to particular occurrences of a periodic group.

The XI option is used for unique descriptors in periodic groups to exclude the occurrence (index) number from the definition of uniqueness.

Because the inverted list requires disk space and update overhead, the descriptor option should be used judiciously, particularly if the file is large and the field that is being considered as a descriptor is updated frequently. For instance, the inverted list for a periodic group used as a descriptor may be very large because each occurrence is stored.

A descriptor may be defined at the time a file is created, or later by using an Adabas utility. Because the definition of a descriptor is independent of and has no effect on the record structure, descriptors may be created or deleted at any time without the need for database restructuring or reorganization.

Note, however, that if a descriptor field is not ordered first in the record structure and logically falls past the end of the physical record, the inverted list entry for that record is not generated for performance reasons. To generate the inverted list entry in this case, it is necessary to unload short, decompress, and reload the file; or use an application program to reorder the field first for each record of the file.

A portion of a field may be defined as a "subdescriptor"; combinations of fields or portions thereof may be defined as a "superdescriptor"; a user-supplied algorithm may be the basis of a "collation descriptor" or "hyperdescriptor"; and a "sounds-like" encoding algorithm may be the basis of a "phonetic descriptor", which may be customized for specific language requirements. See *Special Field and Descriptor Attributes* for more information.

Data Compression Options FI and NU

Default data compression is described in the section *Compression*. At the field level, additional compression can be specified (null suppression option) or all compression can be disabled (fixed storage option).

"Null suppression" (NU) differs from default compression in that searches on descriptor fields defined with null suppression do *not* return records in which the descriptor field is empty.

Fields defined as "fixed format" (FI) do not include a length byte and are not compressed. This option actually saves storage space for one-byte fields or fields that are nearly always full (e.g., a field containing the social security number).

Encoding Conversion Option NV

Alphanumeric (A) or wide-character (W) format fields with the NV option are processed in the record buffer without being converted to or from the user.

The field has the characteristics of the file encoding; that is, the default blank

- for A fields is always the EBCDIC blank (X'40'), and
- for W fields is always the blank in the file encoding for W format.

The NV option is used for fields containing data that cannot be converted meaningfully or should not be converted because the application expects the data exactly as it is stored.

The field length for NV fields is byte-swapped if the user architecture is byte-swapped.

Long Alpha Option LA

The long alphanumeric (LA) option can only be specified for variable-length alphanumeric or wide-character fields; i.e., "A"- or "W"-format fields having a length of zero. With the LA option, such an alphanumeric or wide-character field can contain a value up to 16,381 bytes long.

An alpha or wide field with the LA option is compressed in the same way as an alpha or wide field without the option. The maximum length that a field with LA option can actually have is restricted by the block size where the compressed record is stored.

MU and PE Options and Field Types

Adabas supports two basic field types: elementary fields and multiple-value fields. An "elementary" field has only one value per record. A "multiple-value" (MU) field can have up to 191 values, or occurrences, in a single record. Each multiple-value field has a "binary occurrence counter" (BOC) that stores the number of occurrences.

A "periodic" (PE) group field defines consecutive fields in the FDT that repeat together in a record. Like the members of a non-periodic group field, PE members immediately follow the PE group field, have a higher level number than the PE field, and can be accessed both individually and as a group. Each PE has a BOC that stores the number of occurrences.

A periodic group may be repeated up to 191 times per record and may contain one or more multiple-value fields. Occurrences or values that are not used require no storage space.

Adabas thus supports four field types:

	Single Value per Record	Multiple Values per Record
Single Field	Elementary	MU
Multiple Fields	Group	PE

The following figure illustrates the four field types in a single record structure.

CUSTOMER NUMBER	FIRST NAME	LAST NAME	PE BOC	MU BOC	STREET	CITY	STATE	ZIP
19811	Laura	Cagnetti	3	1	118 Glade	Erie	PA	16509
<div> <div>Elementary Field</div> <div>Group Field (Name)</div> </div>				2	271 Larue	Cincinnati	OH	45211
					P.O. Box 88			
				2	733 Hall	Easton	PA	19014
					P.O. Box 7			
				<div> <div>Multiple Value Field</div> <div>Periodic Group (Address)</div> </div>				

A PE cannot be nested within another PE. Nesting an MU within a PE, as shown in the figure above, is permitted but complicates programming by introducing a two-dimensional array. It also has implications for data access: when Adabas accesses the periodic group, it returns only the first occurrence of the MU for each occurrence of the PE returned.

The unique characteristic of the periodic group and the reason for choosing the periodic group structure is its ability to maintain the order of occurrences. If a periodic group originally contains three occurrences and the first or second occurrence is later deleted, those occurrences are set to nulls; the third occurrence remains in the third position. This contrasts with the way leading null entries are handled in multiple-value fields. The individual values in a multiple-value field do not retain positional integrity if one of the values is removed.

SQL Compatibility Options NC and NN

Special data definition options are included in Adabas to accommodate Software AG's mainframe Adabas SQL Server (ESQ) and other structured query language (SQL) database query languages that require SQL-compatible null representation.

A field designated with the NC (not counted) option may contain a null value that satisfies the SQL interpretation of a field having no value. An NC field containing a null means that *no field value has been entered*; that is, the field's value is not defined.

This undefined state differs from a null value assigned to a non-NC field for which no value has been specified: a non-NC field's null means the value in the field is either zero or blank, depending on the field's format.

The NN (not null) option can be specified only for NC-defined fields. It indicates that an NC field must always have a value defined; it cannot contain an SQL null. This ensures that the field cannot be left undefined when a record is either created or updated. The field value may be zero or blank, however.

Special Field and Descriptor Attributes

Parent Of

The FDT indicates whether a field is a "parent" field for a collation descriptor, sub/superfield, sub/superdescriptor, hyperdescriptor, or phonetic descriptor as described in the following section.

Special Descriptors

Information about any special fields and descriptors (collation descriptors, subdescriptors, subfields, superdescriptors, superfields, phonetic descriptors, and hyperdescriptors) in the file is maintained in the special descriptor table or "SDT" part of the FDT.

SPECIAL DESCRIPTOR TABLE								
TYPE	I NAME	I LENGTH	I FORMAT	I	OPTIONS	I	STRUCTURE	I
	I	I	I	I		I		I
-----I-----I-----I-----I-----I-----I-----I								
SUPER	I H1	I 4	I B	I DE,NU		I AU (1 - 2)		I
	I	I	I	I		I AV (1 - 2)		I
SUB	I S1	I 4	I A	I DE		I AO (1 - 4)		I
SUPER	I S2	I 26	I A	I DE		I AO (1 - 6)		I
	I	I	I	I		I AE (1 - 20)		I
SUPER	I S3	I 12	I A	I DE,NU,PE		I AR (1 - 3)		I
	I	I	I	I		I AS (1 - 9)		I
	I	I	I	I		I		I
PHON	I PH	I	I	I		I PH =PHON(AE)		I
	I	I	I	I		I		I
COL	I Y1	I 20	I W	I DE		I CDX 8,PA		I
COL	I Y2	I 12	I A	I DE,NU,PE		I CDX 1,AR		I
	I	I	I	I		I		I
	I	I	I	I		I		I
-----I-----I-----I-----I-----I-----I-----I								

Along with the name, length, format, and specified options of each special field and descriptor, this table provides the following information:

Column	Explanation
TYPE	COL Collation descriptor HYPER Hyperdescriptor PHON Phonetic descriptor SUB Subfield/subdescriptor SUPER Superfield/superdescriptor
STRUCTURE	The component fields and field bytes of the sub-, super-, or hyperdescriptor. Phonetic descriptors show the equivalent alphanumeric elementary fields. Collation descriptors show the associated collation descriptor userexit and the name of the parent field.

Collation Descriptor

An alphanumeric or wide-character field can be defined as a parent field of a "collation" descriptor. A collation descriptor is used to sort field values in a special user-defined sequence. The LF command reports the collation descriptor field information.

A collation descriptor is assigned a collation descriptor user exit (1-8) which encodes the collation descriptor value and decodes it back to the original field value. The ADARUN parameter CDXnn is used to specify collation descriptor user exits.

Hyperdescriptor

The hyperdescriptor option can be used to generate descriptor values based on a user-supplied algorithm. Up to 31 different hyperdescriptors can be defined for a single physical Adabas database. Each hyperdescriptor must be named by an appropriate HEXnn ADARUN statement parameter in the job where it is used.

With hyperdescriptors, "fuzzy" matching is possible; i.e., retrieving data based on *similar* rather than on *exact* search criteria. Hyperdescriptors allow multiple virtual indexes, meaning that several different search index entries can be made for a single data field.

Hyperdescriptors can be used to implement "n"-component superdescriptors, derived keys, or other key constructs. Using hyperdescriptors, it is possible to develop applications that are simpler and more flexible than applications based on a strictly normalized relational structure.

One application area for hyperdescriptors is name processing. For example, the name SCHROEDER could be stored not only with the index SCHROEDER itself, but also with the "virtual" indexes SCHRODER, SCHRADER, SCHR�DER or any other variation of the name. Thus, although only the name SCHROEDER is physically stored in the data area of the database, multiple search indexes exist to the data. If, subsequently, a search is made for the name SCHRODER, the record SCHROEDER will be found.

A more sophisticated application area for hyperdescriptors is fingerprint matching, in which typical characteristics of fingerprints can form the basis of a fuzzy matching algorithm; i.e., the original fingerprint is stored in the database, but any number of search indexes can be made to the fingerprint, based on an algorithm that allows small-scale deviations from the original.

Phonetic Descriptor

A "phonetic" descriptor may be defined and used to search for all records that contain similar phonetic values. The phonetic value of a descriptor is determined by an internal algorithm based on the first 20 bytes of the field value with only alphabetic values being considered (numeric values, special characters and blanks are ignored).

Subfield / Superfield

A portion of a field ("subfield") or any combination of fields ("superfield") may be defined as an elementary field (see *MU and PE Options and Field Types*). Subfields and superfields may be used for read operations only. They may only be changed by updating the original fields.

Subdescriptor

A "subdescriptor" is part of a single field used as a descriptor. The field from which the subdescriptor is derived may or may not be an elementary descriptor (see *Descriptor Options DE, UQ, and XI*). If a search criteria involves a range of values contained in the first "n" bytes of an alphanumeric field or the last "n" bytes of a numeric field, a subdescriptor may be defined using only the relevant bytes of the field. A subdescriptor allows you to increase the efficiency of a search by specifying a single value rather than a range of values.

For example, if the first two bytes of a five-byte field refer to a geographical region and you want to retrieve all records for region 11 without using a subdescriptor, you would have to search for all records in the range 11000-11999. If you define a subdescriptor comprising the first two bytes of the field, you could search for all records with 11 in the subdescriptor.

Superdescriptor

A "superdescriptor" combines all or parts of 2-20 fields. The fields from which the superdescriptor is derived may or may not be elementary descriptors. When search criteria involve values for a combination of fields, using a superdescriptor is more efficient than using a combination of several elementary descriptors.

For example, to search for customers by last name within regions, you could create a superdescriptor by combining the first two bytes (i.e., the geographical region indicator) of the five-byte customer number field and the entire customer last name field.